Relatório de Trabalho: Estrutura de Dados I

Participantes: Eike Sousa Cangussú (2024203418), Gabriel Meireles Amaral (2024201615),Matheus Alves Martins(2024201612), Guilherme Lima dos Anjos(2021201054)
Data:17/08/25

Visão Geral do Trabalho

Neste trabalho foram usados os conhecimentos adquiridos em sala de aula, para implementar uma estrutura de dados pilha usando tipo genérico de dados, e, a partir desta, buscar soluções para três problemas propostos. O uso do tipo genérico foi ideal, para permitir o reaproveitamento do código da estrutura para os tipos necessários em cada problema.

Implementação da Pilha Genérica

- **Definição da Estrutura:** Na implementação do tipo abstrato de dados, o caminho escolhido foi usar uma lista simplesmente encadeada para representar os elementos da pilha, dentro do que se mostraram como as necessidades para a resolução dos problemas, foi uma escolha acertada. A opção pela lista simplesmente encadeada se deu pela simplicidade da implementação e a total usabilidade para as funções do tad pilha, que exigem manipulação das chaves somente do topo, como a lista simplesmente encadeada nos dá acesso direto somente ao primeiro elemento da lista, tem tudo que precisamos para a implementação da mesma de forma satisfatória. A estrutura Pilha definida possui os seguintes campos:
    - No *topo: Carrega o nó para o topo da lista encadeada que define a pilha, esse nó é uma estrutura que possui os campos:
        - void *ch: Para representar a chave guardada pelo nó.
        - No *prox: Aponta para o elemento abaixo do nó na pilha.

    - int tamanho: Tipo int carrega o número de elementos presentes na pilha.

  OBS: Para auxiliar na alocação de memória para a estrutura Nó foi criada a função No *criaNo(void *chave) que aloca dinamicamente a memória e preenche os campos de um nó.
- **Funções essenciais:** Dentro do TAD pilha, existem algumas funções essenciais que são as operações de uma pilha, por ser uma função de propriedade LIFO “Last-in, First-out” temos a função
    - desempilha(): Recebe uma pilha remove o elemento presente no seu topo, se existir e o retorna.
    - empilha(): Recebe uma pilha e uma chave do tipo void* aloca um novo nó e adiciona ele ao topo da estrutura.
    - esvazia() : Recebe uma pilha remove os elementos do topo e libera a memória até que a lista esteja vazia.
    - vazia() : Recebe uma pilha e verifica se ela está vazia ou possui elementos.
    - tamanho() : Recebe uma pilha e retorna o seu tamanho, ou seja, o número de elementos que foram empilhados na estrutura.
    - iniciaPilha() : Não recebe nada como parâmetro, cria internamente da função uma pilha, aloca a memória para a mesma e retorna essa pilha alocada.
    - destroiPilha() : Recebe uma pilha, verifica se ela está vazia, depois se sua lista de elementos está vazia, se não estiver, chama a função esvazia(), depois libera a memória da pilha recebida.
- **Funções extras:**
    - mostrarTopo() : Recebe uma pilha e retorna a chave do seu topo, sem necessariamente desempilha-la. Retorna NULL caso o topo seja NULL.
- **Generidade:** Toda a estrutura foi implementada sem o uso de um tipo específico de dados, como inteiro, caractere ou float. Em vez disso, foi utilizado um ponteiro para o tipo void, o que permite, com os devidos cuidados, que sejam atribuídos a ele endereços de memória para diferentes tipos de dados. Isso torna a implementação dessa pilha de uso variado para diversas soluções. Ao utilizar a estrutura, o usuário deve se atentar a inserir somente ponteiros para endereços do tipo que deseja usar, e ao precisar receber chaves provenientes da estrutura, deve fazer o casting do retorno para o tipo que está utilizando.

- **Considerações finais sobre a estrutura:** Dentro do que se pede a implementação de uma pilha genérica, foram dadas ao usuário, por meio das funções implementadas, todas as ferramentas necessárias para a correta administração da memória, garantindo que, com o uso correto, todas as estruturas sejam devidamente alocadas e desalocadas. Com esse formato, a estrutura se mostrou útil e eficiente para a resolução dos problemas propostos.

Resolução dos Problemas

Problema 1:

- **Descrição do Problema:**

  Nesse problema, devemos implementar um algoritmo que faça validação de expressões matemáticas digitadas pelo usuário. Utilizando a implementação do TAD Pilha para tipo genérico de dados. As expressões devem conter literais de A a J, operadores (+ (adição), - (subtração), / (divisão),* (multiplicação), ^ (potenciação)) e delimitadores de escopo (“(”, “)”, “[”, “]”, “{”, “}”). O programa deve prever 2 tipos de validação:

  a) Expressões contendo parênteses, colchetes e chaves, apenas verificando se cada delimitador aberto possui seu fechamento realizado corretamente.

  b) Expressões contendo parênteses, colchetes e chaves, considerando a precedência entre os delimitadores: os escopos devem ser abertos e fechados corretamente e deve-se verificar se os escopos dos colchetes abrangem os parênteses e os escopos das chaves abrangem os colchetes.

- **Solução com Pilha:**

  A pilha opera como um sistema de controle dos delimitadores abertos na expressão. Quando o algoritmo inicia, a pilha começa vazia, pronta para adicionar aberturas. Cada vez que um delimitador de abertura, como (, [ou { , é encontrado durante a leitura da expressão, ele é imediatamente adicionado na

pilha. Essa inserção funciona como um lembrete de que aquela abertura precisa ser fechada posteriormente, mantendo a ordem exata em que os delimitadores apareceram.

Quando o algoritmo identifica um delimitador de fechamento, como),] ou }, ele consulta a pilha para verificar o último registro de abertura. Se a pilha estiver vazia nesse momento, significa que não há abertura correspondente para aquele fechamento, e a expressão é invalidada. Se a pilha ainda contém registros, o último delimitador aberto é recuperado e comparado com o fechamento atual. Essa comparação verifica se os tipos são correspondentes: parênteses com parênteses, colchetes com colchetes, chaves com chaves. Se não forem correspondentes, a expressão é considerada inválida.

Após toda a expressão ser lida, o algoritmo realiza uma verificação final na pilha. Se ainda houver registros de abertura não fechados, invalida a expressão.

- **Algoritmo:**

  O algoritmo implementa duas funções centrais para a validação das expressões: validaExpTipoA() e validaExpTipoB(). Ambas compartilham uma estrutura lógica similar, porém a segunda função adiciona verificações para garantir o cumprimento das regras de precedência entre os delimitadores. As funções recebem como parâmetro um vetor de caracteres representando a expressão matemática e utilizam uma variável ‘flag’ para controlar o estado de validade, iniciada com valor 1 (válida) e modificada para 0 (inválida) caso sejam detectadas violações durante a análise.

  A função validaExpTipoA() inicia criando uma pilha através de iniciaPilha() e define a ‘flag’ de validação como 1. O processo de análise percorre cada caractere da expressão sequencialmente. Quando encontra um delimitador de abertura, como '(', '[' ou '{', ele aloca memória dinamicamente para armazenar o caractere e empilha esse elemento. Ao detectar um delimitador de fechamento, como ')', ']' ou '}', verifica primeiro se a pilha está vazia. Caso esteja, altera a ‘flag’ para 0, indicando um fechamento sem abertura

correspondente. Se a pilha contiver elementos, desempilha o topo e compara com o fechamento atual. Se os caracteres não corresponderem (como '(' com ']'), altera a ‘flag’ para 0 e interrompe a análise. Após percorrer toda a expressão, realiza uma verificação final: se a pilha não estiver vazia, define a flag como 0, sinalizando aberturas pendentes de fechamento. Antes de retornar o resultado, a função libera toda memória alocada e destrói a estrutura da pilha.

A função validaExpTipoB() segue o mesmo fluxo básico, mas adiciona uma etapa crítica antes do empilhamento. Ao identificar um delimitador de abertura, utiliza a função topo() para inspecionar o elemento superior da pilha sem removê-lo. Essa observação permite verificar violações na precedência: se um colchete '[' for aberto sobre um parêntese '(', ou se uma chave '{' for aberta sobre parêntese '(' ou colchete '[', configura imediatamente a ‘flag’ como 0 e interrompe o processo. Essa verificação adicional assegura que colchetes nunca estejam contidos dentro de parênteses, e chaves nunca estejam dentro de parênteses ou colchetes. O tratamento de fechamentos e a verificação final de elementos residuais na pilha mantêm-se idênticos à função anterior. O gerenciamento de memória com liberação de elementos desempilhados e destruição da pilha no final garante a ausência de vazamentos e uso de memória desnecessários.

A diferença essencial entre as funções reside na camada de validação hierárquica presente apenas em validaExpTipoB(). Enquanto a primeira função foca exclusivamente no pareamento correto de aberturas e fechamentos, a segunda impõe restrições adicionais sobre a relação entre os tipos de delimitadores, exigindo que escopos maiores (chaves) envolvam escopos menores (colchetes), que por sua vez devem envolver os menores (parênteses). Ambas preservam a eficiência do algoritmo com complexidade linear O(n), onde *n* é o comprimento da expressão, e compartilham o mesmo mecanismo de controle através da variável `flag` para comunicar o resultado da validação ao código chamador.

Por fim, implementamos a função main para controlar as entradas e saídas, recebendo a expressão iniciando as funções e imprimindo na tela os seus resultados.

- **Resultados:**

Após concluir a implementação do algoritmo, realizamos uma série de testes para verificar cada cenário de saída. No primeiro teste, utilizamos a expressão [a + b * (d - c)]. O resultado obtido confirmou a validação para ambos os tipos:
Saída: A expressão [a + b * (d - c)] é validada pelas opções (a) e (b).
Esse resultado era esperado, pois a expressão mantém o balanceamento dos delimitadores e respeita a hierarquia de escopos (colchetes externos e parênteses internos).

No segundo teste, inserimos uma expressão propositalmente incorreta: a * j - (c + e]. O algoritmo identificou corretamente os erros:
Saída: A expressão a * j - (c + e] não é validada em nenhuma das opções.
A invalidação ocorreu devido a dois problemas críticos: um parêntese aberto sem fechamento correspondente e um colchete fechado sem abertura prévia, violando as regras básicas de ambos os tipos.

Para o terceiro teste, selecionamos a expressão ({a / b} * [(e - f)]), válida apenas para o tipo A. O resultado refletiu precisamente essa condição:
Saída: A expressão ({a / b} * [(e - f)]) é validada apenas pela opção (a).
A validação falhou para o tipo B porque a expressão viola a precedência hierárquica: os colchetes [...] estão dentro de chaves {...}, mas os parênteses (...) estão dentro de colchetes, criando uma estrutura onde escopos menores (parênteses) estão contidos em escopos intermediários (colchetes), o que é permitido apenas se os colchetes estiverem dentro de chaves..

Problema 2:

- Descrição do Problema: Este problema visa criar um programa para converter expressões matemáticas da notação infixa para posfixa. A notação infixa (A+B) posiciona o operador entre os operandos, enquanto a posfixa o coloca após, eliminando a necessidade de parênteses para precedência. O programa deve

suportar variáveis literais (A-J), operadores (+, -, /, *, ^) e parênteses, com uma interface de menu para escolha do formato de entrada.

- Solução com Pilha: A pilha genérica é essencial para a conversão infixa-posfixa. Ela gerencia operadores e parênteses de abertura, garantindo a ordem correta de precedência e associatividade. Operandos são diretamente adicionados à saída posfixa. Operadores são empilhados ou desempilhados para a saída com base em sua precedência e associatividade, enquanto parênteses controlam o agrupamento das operações. Ao final, a pilha é esvaziada para a saída.
- Algoritmo: A função converteInfixa processa a expressão infixa caractere por caractere:

  Variáveis Literais (A-J): Adicionadas diretamente à string posfixa.

  Parênteses de Abertura (: Empilhados.

  Operadores: Sua precedência é comparada com o topo da pilha. Operadores de maior ou igual precedência no topo são desempilhados para a saída antes que o operador atual seja empilhado. Exceção: ^ (potenciação) tem associatividade à direita.

  Parênteses de Fechamento): Operadores são desempilhados para a saída até encontrar o (correspondente, que é removido da pilha.

  A função main() oferece um menu para o usuário escolher o formato de entrada. A opção de entrada infixa chama converteInfixa para realizar a conversão, incluindo uma validação para balanceamento de parênteses. A opção de entrada posfixa atualmente apenas exibe a string de entrada, não realizando a avaliação.

  - Resultados: A funcionalidade de conversão de infixa para posfixa está implementada com sucesso, processando corretamente a precedência e associatividade. Exemplos incluem:

    A+B*C → ABC*+

    (A+B)*C → AB+C*

A^B^C → ABC^^

Problema 3:

- **Descrição do Problema:** Neste problema devemos implementar um algoritmo que diga quantos cômodos existem dentro de uma planta de uma casa , representada por uma matriz n x m quadrados , onde cada quadrado pode ser uma parede(“ # ”) ou um piso(“ . ”).Sendo um cômodo definido por uma região conexa de pisos , onde somente é possível transitar entre eles , pelas 4 direções (esquerda, direita, cima, baixo).

- **Solução com Pilha:** A pilha foi utilizada como mecanismo central para explorar os pisos conectados. Cada posição da matriz que representa um piso é modelada por uma estrutura
  Posicao (x, y), e sempre que encontramos um novo piso não visitado, sua coordenada é empilhada para analisarmos posteriormente. Onde o funcionamento ocorre da seguinte forma :
  Empilha a posição inicial de um piso quando ele é encontrado;
  Enquanto a pilha não estiver vazia , desempilha a posição atual , verificando seus vizinhos(cima, baixo, esquerda e direita), para cada vizinho que também for um piso, o marca como visitado e o empilha.O processo só termina quando a pilha estiver vazia, indicando que o cômodo foi percorrido.
  A pilha atua como um controlador de estruturas, substituindo a recursão tradicional da busca por profundidade.

- **Algoritmo:**
  O algoritmo implementa uma busca por profundidade utilizando TAD pilha para tipo genéricos de dados, onde lê dois números inteiros n e m que representam a linha e coluna da planta ,juntamente com a matriz de caracteres que a formam.

  A estrutura Posicao é utilizada para armazenar as coordenadas de cada célula da matriz, permitindo que a posição seja tratada como um objeto genérico que pode ser empilhado e desempilhado, garantindo flexibilidade e controle durante

a exploração. Cada posição contém dois campos(x , y) que representam respectivamente a linha e a coluna da célula no mapa.

A função buscaComodo é responsável por explorar completamente um cômodo a partir de uma posição inicial de piso, ela recebe como parâmetros as dimensões da matriz, a posição inicial.A função cria uma pilha genérica, empilha a posição inicial e marca essa célula como visitada. Em seguida, enquanto a pilha não estiver vazia, desempilha-se a posição do topo e verifica-se cada vizinho nas quatro direções possíveis: cima, baixo, esquerda e direita. Caso um vizinho represente um piso e ainda não tenha sido visitado, ele é empilhado e marcado. Esse processo garante que toda a região conectada seja percorrida, identificando corretamente o cômodo,ao final, a pilha e a memória alocada para as posições são liberadas, garantindo eficiência e evitando vazamentos de memória.

A função main gerencia a execução do programa, nela realiza a leitura e validação das dimensões da matriz, inicializa as matrizes de planta e controle e percorre todas as posições da matriz. Sempre que encontra um piso não visitado, chama a função buscaComodo e incrementa o contador de cômodos,após a varredura completa, o programa imprime o número total de cômodos encontrados.

As variáveis auxiliares dx e dy são utilizadas para simplificar a navegação pelas quatro direções possíveis, permitindo que o algoritmo verifique os vizinhos de forma organizada e eficiente. O uso do TAD Pilha para tipos genéricos permite controlar dinamicamente quais posições ainda precisam ser exploradas, substituindo a recursão direta e garantindo que o programa funcione corretamente mesmo com matrizes de grandes dimensões.

Em síntese, a funcionalidade do código é baseada na exploração iterativa de cada cômodo por meio da função buscaComodo, utilizando a estrutura Posicao para representar coordenadas, a pilha genérica para gerenciar posições pendentes e as matrizes de controle e mapa para armazenar o estado da exploração. A função main coordena a leitura, a validação e a contagem dos cômodos, garantindo a execução correta e eficiente do programa.

- **Resultados:**

A precisão do resultado do código após toda leitura , identificação e contagem dos cômodos está diretamente ligada ao correto funcionamento das matrizes auxiliares:

A matriz que armazena o mapa(planta) garante a fidelidade da representação do ambiente, enquanto a matriz de controle de pisos visitados impede repetições e assegura que cada posição seja analisada apenas uma vez,além disso, o uso da pilha garante que a exploração dos cômodos seja completa, independentemente do tamanho da planta ou da complexidade do desenho.

Em termos práticos, os resultados obtidos mostram que o programa é capaz de distinguir cômodos em plantas simples ou complexas, identificando até múltiplos cômodos separados por paredes. Dessa forma, o algoritmo comprova sua eficiência ao fornecer, de maneira automatizada e confiável, a contagem correta do número de cômodos em qualquer planta válida fornecida como entrada.

Documentação dos Códigos

- **Tad pilha:**

  Função: criaNo
  Descrição: Aloca dinamicamente uma estrutura No e preenche seus campos.
  Entrada: void *chave: ponteiro para a chave de informação de tipo genérico a ser guardada no nó.
  Saída: Ponteiro para estrutura No alocada dinamicamente.
  Declaração: No *criaNo(void *chave);

  Função: esvazia
  Descrição: Esvazia a pilha removendo todos os seus elementos e ajustando o tamanho.
  Entrada: Pilha *p: Pilha a ser esvaziada.
  Saída: Ponteiro para estrutura Pilha esvaziada.
  Declaração: Pilha *esvazia(Pilha *p);

  Função: vazia
  Descrição: Verifica se a Pilha está vazia e retorna o resultado
  Entrada: Pilha *p: Ponteiro para pilha a ser verificada.
  Saída: Valor booleano para identificar se a estrutura está vazia ou não.

Declaração: bool vazia(Pilha *p);

Função: empilha
Descrição: Cria um nó com o dado recebido, adiciona um novo elemento no topo da pilha, atualiza o novo topo e aumenta o tamanho.
Entrada: Pilha *p: Ponteiro para a pilha que receberá um novo elemento no seu topo.
void *novo: Ponteiro para dado de tipo genérico a ser inserido na pilha.
Saída: void.
Declaração: void empilha(Pilha *p, void *chave);

Função: desempilha
Descrição: Exclui o elemento que está no topo da pilha, decrementa em um o tamanho, atualiza o novo topo e retorna o elemento removido.
Entrada: Pilha *p: Ponteiro para a pilha a que terá um elemento excluído do seu topo.
Saída: void * Elemento removido da pilha ou NULL se a pilha já estiver vazia.
Declaração:void *desempilha(Pilha *p);

Função: tamanho
Descrição: Identifica e retorna o número de elementos em uma pilha.
Entrada: Pilha *p: Ponteiro para a pilha a ter seu tamanho checado.
Saída: Inteiro com o número de elementos da pilha.
Declaração:int tamanho(Pilha *p);

Função: iniciaPilha
Descrição: Inicia uma pilha vazia.
Entrada: void
Saída: Ponteiro para pilha inicializada com tamanho = 0, topo = NULL
Declaração: Pilha *iniciaPilha();

Função: destroiPilha(Pilha *P)
Descrição: Esvazia e destrói completamente uma pilha.
Entrada: Ponteiro para a pilha a ser destruída
Saida: void
Declaração: void destroiPilha(Pilha *p);

Função: mostrarTopo(Pilha *P)
Descrição: Retorna o topo de uma pilha sem que ela seja desempilhada, ou NULL caso a pilha esteja vazia
Entrada: Ponteiro para a pilha que terá o topo exibido
Saida: void * Chave do elemento no topo da pilha ou NULL caso estiver vazia.
Declaração: void *mostrarTopo(Pilha *p);

- **Problema 1:**

  Função: validaExpTipoA
  Descrição: Vai receber um vetor de caracteres com a expressão de entrada e retorna se a expressão é valida ou não.
  Entrada: char expressão[], um vetor de caracteres com a expressão matemática que o usuário colocar como entrar.
  Saída: Retorna o valor 1 ou 0, representando a validação ou não da expressão do tipo A.
  Declaração: int validaExpTipoA(expressao);

  Função: validaExpTipoB
  Descrição: Vai receber um vetor de caracteres com a expressão de entrada e retorna se a expressão é valida ou não.
  Entrada: char expressão[], um vetor de caracteres com a expressão matemática que o usuário colocar como entrar.
  Saída: Retorna o valor 1 ou 0, representando a validação ou não da expressão do tipo B.
  Declaração: int validaExpTipoB(expressao);

Problema 2:

  Função expressaoValida
  Descrição: Verifica se possui o balanceamento correto de parênteses.
  Entrada: char *infixa: Ponteiro para a string que contém a expressão infixa.
  Saída: bool: Retorna true se os parênteses da expressão estão balanceados, false caso contrário.
  Declaração: bool expressaoValida(char *infixa);

  Função: entradavalida
  Descrição: Valida se um caractere específico é um tipo de entrada permitido para as expressões matemáticas.
  Entrada: char exp: O caractere a ser verificado.
  Saída: bool: Retorna true se o caractere é uma entrada válida, false caso contrário.
  Declaração: bool entradavalida(char exp);

  Função: operador
  Descrição: Verifica se o caractere fornecido é um dos operadores matemáticos reconhecidos pelo programa (+, -, /, *, ^).
  Entrada: char exp: O caractere a ser avaliado.
  Saída: bool: Retorna true se o caractere é um operador, false caso contrário.
  Declaração: bool operador(char exp);

Função: precedencia
Descrição: Retorna um valor inteiro que representa a precedência de um operador matemático.
Entrada: char operador: O caractere do operador.
Saída: int: Um inteiro que indica o nível de precedência do operador. Retorna 0 para caracteres não reconhecidos como operadores.
Declaração: int precedencia(char operador);

Função: converteInfixa
Descrição: Converte uma expressão matemática do formato infixo para o formato posfixa (ou Notação Polonesa Reversa)
Entrada: char *infixa: Ponteiro para a string que contém a expressão infixa a ser convertida.
Saída: void (não retorna valor diretamente; imprime a saída).
Declaração: void converteInfixa(char *infixa);

Função: main
Descrição: É a função principal, gerencia o menu de interação com o usuário, permitindo escolher entre inserir uma expressão no formato posfixa ou infixa.
Entrada: void (não recebe parâmetros).
Saída: int: Retorna 0 para indicar que o programa terminou com sucesso.
Declaração: int main();

- **Problema 3:**

Função: buscaComodo
Descrição: Função de busca dos cômodos recebendo como parâmetro o tamanho de linha e coluna(n, m) da matriz do mapa (planta) e a posição inicial(i , j) para identificar onde começar a busca.
Entrada: int n , m : Tamanho de linha e coluna da matriz;
int i , j : Posição inicial para início da busca
Saída: void (não retorna nada diretamente , apenas atualiza a matriz visitou, matriz de posições visitadas)
Declaração: void buscaComodo(int n , int m , int i , int j)

Função: main
Descrição: Organiza a leitura dos dados, valida as entradas, inicializa as estruturas auxiliares e um contador de cômodos, chama a função de busca para identificação dos cômodos e, por fim, exibe o resultado ao usuário.
Entrada: void (não recebe parâmetros)
Saída: Retorna 0 , para indicar que o programa encerrou com sucesso;
Declaração: int main()